ChatGPT

Migration Plan for AWS Containerization & HITRUST Compliance

AWS Service Mapping & Architecture

Compute (ECS) – We will containerize the HealthPrep application and deploy it on **AWS ECS**, preferably using Fargate for a serverless container approach [1]. This provides full control without managing EC2 instances and is cost-effective at our scale. The containerized app will run behind an **Application Load Balancer (ALB)** to handle inbound HTTPS traffic. The ALB will reside in public subnets, forwarding requests to ECS tasks in private subnets [2]. This setup ensures the app is accessible securely while the backend compute remains isolated.

Database (PostgreSQL) – Instead of the current Neon PostgreSQL, we plan to use **Amazon RDS for PostgreSQL** in AWS. RDS will store our application data with encryption at rest enabled (AES-256) and offer automated backups and point-in-time recovery [3]. The RDS instance will be placed in the private subnets of our VPC with no public exposure [4]. (Neon is hosted on AWS, but migrating to RDS under our account provides direct control, inclusion in our AWS **Business Associate Agreement (BAA)**, and easier compliance reporting.) RDS snapshots and AWS Backup will be used for backups: daily snapshots retained per policy and/or daily logical dumps to S3 for long-term archival [5].

File Storage – Currently, PHI-redacted document transcripts are stored in `/admin/documents` locally. In AWS, we will use **Amazon S3** for this purpose. All uploaded or generated documents (e.g. OCR results, prep sheet files if any) will be stored in an S3 bucket. S3 provides durable storage with server-side encryption and lifecycle policies (we can configure automatic transition to Glacier for long-term retention) [6] [3]. The application will be updated to read/write transcripts from S3 instead of local disk. This ensures persistence beyond container lifecycle and centralizes PHI storage with access controls and logging. We will enable S3 access logging and bucket policies to restrict access to only our application roles.

Secrets & Configuration – All secrets (API keys, database URL, encryption keys, etc.) will be managed by **AWS Secrets Manager**. We'll create a structured hierarchy (e.g. `/healthprep/prod/...`) for secrets as outlined in our Key Management Policy [7]. For example, `SECRET_KEY` (Flask session secret) might be stored at `/healthprep/prod/secret-key`, and the Epic credentials at `/healthprep/prod/epic/...`. The ECS task definition will reference these secrets so that they are injected as environment variables at runtime [8]. Secrets Manager encrypts values with AWS KMS and provides audit logging via CloudTrail [9]. This approach mirrors our current use of Replit's secret manager but with more robust control and rotation features.

Logging & Monitoring – We will configure **Amazon CloudWatch** for application and infrastructure logs. The Flask app's logs (and any audit logs we emit) will be sent to CloudWatch log groups. We will set log retention to meet our compliance needs (e.g. 7 years for audit logs) [10] by using S3 or Glacier for archival if needed. CloudWatch metrics and alarms will be set up for key indicators (error rates, CPU usage, etc.). Additionally, we will enable **AWS CloudTrail** across the account to log all AWS API activity, which is critical for security auditing and HITRUST compliance [11]. CloudTrail logs will help us monitor changes to resources, especially security-relevant actions like secret access or configuration changes.

Network (VPC Design) – We'll deploy everything into an **AWS Virtual Private Cloud (VPC)** to isolate our system. The VPC will have both public and private subnets: the ALB lives in the public subnet to receive traffic, whereas ECS tasks, the RDS database, and any Redis/ElastiCache instance will reside in private subnets with no direct internet access [2]. For ECS tasks to call external services (e.g. Epic's API endpoints), we will route outbound traffic through a **NAT Gateway** in the public subnet [12]. Security Groups will enforce least access: the ALB's security group allows inbound HTTPS (443) from the internet and only sends traffic to the ECS security group [13]. The ECS tasks' security group permits inbound from the ALB and outbound only to necessary endpoints (e.g. HTTPS to Epic, port 5432 to the RDS's SG) [4]. The RDS security group allows incoming connections only from the ECS tasks [14]. This design (which we will document in an architecture diagram) ensures a tightly controlled network, satisfying AWS best practices and audit requirements.

Additional AWS Services – We will leverage other services as needed: for example, **AWS KMS** for managing encryption keys (KMS underpins S3, RDS, and Secrets Manager encryption by default, and we can use a CMK if required for compliance). If our application needs caching or background task processing, we might introduce **Amazon ElastiCache (Redis)** for caching or **Amazon SQS/ SNS** for queuing tasks. The deployment docs suggest using Redis for background jobs [15] – if we have long-running OCR tasks, we could offload them to a queue and have separate ECS workers or AWS Lambda consume them. This isn't strictly required for migration, but it's an option to improve scalability and control costs (e.g., we could spin up separate ECS tasks for batch OCR processing on demand).

Adapting Application Components for AWS

Dockerization of the App – We will create a `Dockerfile` for HealthPrep that installs all necessary system packages and Python dependencies (Flask, PyMuPDF, Tesseract OCR, etc.). This ensures the runtime environment on ECS matches what we had on Replit. We'll harden the image (using a slim base image, applying OS updates, and excluding dev tools) as guided by our system hardening standards. Once the Dockerfile is ready, we can test the container locally, then push it to AWS ECR for deployment.

Environment Configuration – The app is already designed to load config from environment variables (the Deployment Readiness report confirms **"zero hardcoded secrets – all configuration via env variables"** [16] [17]). We'll carry this forward by populating those env vars through ECS task definition secrets. Our `.env.example` can serve as a reference for what needs to be set. In AWS, we won't keep a `.env` file; instead, the ECS task inherits values from Secrets Manager (e.g., the task JSON will have `SECRET_KEY` valueFrom the Secrets Manager ARN) [8]. We will verify that the `utils/secrets_validator.py` still passes – it will run at startup and ensure all required secrets are present [18].

Persistent Document Storage – For transcripts currently saved under `/admin/documents`, we need to refactor that part of the application. Options include mounting a network file system (**EFS**) to ECS or using **S3**. We prefer S3 for its durability and audit features. That means, for example, after OCR is performed on a document, the resulting text (with PHI redacted) would be uploaded to an S3 bucket rather than saved to the container's filesystem. Later, when users or processes need that transcript, the app can fetch it from S3. We'll use AWS SDK (boto3) in our code or possibly presigned URLs if front-end access is needed. This change will require updating file I/O code in the app, but it aligns with our “minimal data storage” principle since S3 can be configured to expire or archive data as needed. Moreover, storing documents in S3 helps with compliance (easy to encrypt and audit access). We will ensure that **PHI remains redacted** as it is now – our pipeline of “OCR -> PHI filter -> storage” remains, just the storage location changes [19] [20]. The database will continue to store references to these documents (and any metadata), as it does currently.

Application Logging – We will adjust logging to suit AWS. In a container on ECS, writing to stdout/stderr will send log streams to CloudWatch automatically. We’ve already audited logs to ensure no PHI or secrets leak into them (this was part of readiness hardening) [21] [22]. We’ll set the logging format to include context like timestamp, org/user, etc., if not already. Also, the **audit logs** the app maintains (e.g., AdminLog entries for user actions, document access logs, etc.) will remain in the database as a source of truth, but we might also export or mirror critical security events to CloudWatch for real-time monitoring. A CloudWatch metric filter could, for example, watch for “PHI filter failure” alerts or multiple failed login attempts (since our security alerts code already handles these events) [23] [24]. In AWS, we can set up alarms or SNS notifications for such patterns.

Database Connection – If moving to RDS, we’ll update the `DATABASE_URL` in Secrets Manager to the new RDS endpoint. Our application likely uses SQLAlchemy with the URL from env, so this should be straightforward. We must also update any security group or connection info to allow the ECS tasks to connect to RDS. One consideration: Neon (the current DB) might have slightly different performance or extensions; we should ensure the RDS instance is configured equivalently (e.g., if we used any specific Postgres extensions or settings). We’ll also enable **IAM DB authentication** for RDS if desired (this would remove the need to store a DB password by using IAM roles, but since we already have a DB URL secret, this is optional). Regardless, we will enforce SSL connections to RDS (parameter `sslmode=require` in the DATABASE_URL) to encrypt data in transit.

Third-Party Integrations – All external calls (e.g., to Epic’s FHIR API, to Stripe or Resend if those exist for payments or emails) will now originate from our AWS environment. With the NAT Gateway in place, these calls should work, but we might need to update allowed callback URLs or IP ranges on those services. For Epic specifically, calls to Epic’s endpoints will come from our AWS egress IPs; if Epic whitelists IPs, we might need to provide them the NAT gateway IP. Stripe/Resend likely use API keys and don’t require IP allow-listing, but we will double-check any such requirements during the migration.

Cost Management – We are conscious of runaway compute costs for OCR and bulk processing. On AWS, we will mitigate this by: using **ECS task auto-scaling** (scale out when needed, scale in when idle), and possibly offloading OCR heavy-lifting to separate on-demand tasks or AWS Batch. We will monitor resource usage via CloudWatch. Additionally, we can set AWS Budgets/Alarms to get notified if costs spike unexpectedly. Since the main costs are document ingestion and sheet generation, we can also throttle these processes (for example, limit the number of concurrent OCR tasks, or schedule bulk prep sheet jobs during off-peak times). By having these tasks in our control (possibly in a queue), we avoid unbounded parallelism that could rack up cost. The team will have CloudWatch dashboards to observe these workloads. Overall, AWS gives us tools to control scaling, which should help us keep compute costs predictable while still handling necessary workload.

HITRUST i1 Compliance & Security Considerations

Shared Responsibility & Inherited Controls – Moving to AWS helps with HITRUST i1 because AWS's infrastructure is already certified for many controls. AWS provides a *Shared Responsibility Matrix* for HITRUST, which delineates what AWS covers vs. what we must cover. For example, AWS handles physical data center security, network infrastructure security, and underlying hypervisor patching – those controls we **inherit** from AWS [11]. Our responsibility is to configure our cloud resources securely and manage application-level controls. We need to implement all HITRUST CSF controls that apply to our application and data. In practice, this means using AWS services in a secure manner (enabling encryption, using IAM, monitoring, etc.) and maintaining our policies/procedures around them. We will obtain the latest AWS HITRUST Shared Responsibility Matrix to ensure we haven't missed any inherited controls or those we must implement.

AWS Business Associate Agreement (BAA) – Before launching PHI on AWS, we will sign a BAA with AWS for HIPAA compliance [25]. This legal agreement is required and will cover services like EC2, S3, RDS, etc. (basically all the ones we use that store or process PHI). Fortunately, AWS's list of HIPAA-eligible services includes ECS, RDS, S3, Secrets Manager, etc., so we're within bounds. The BAA ensures AWS takes on responsibility for their part of security (and breach notification on their side), aligning with HITRUST requirements.

Encryption Everywhere – We will enforce **encryption at rest and in transit** for all components, as reflected in our policies. RDS data is encrypted at rest (and snapshots too) [26]. S3 will have default encryption (SSE-S3 or SSE-KMS) for all buckets, and we can even limit that only TLS connections can retrieve objects. Secrets Manager and KMS handle encryption of secrets/keys by default [9]. In transit, the ALB will enforce HTTPS only – our app already has HTTP->HTTPS redirect logic for production [27] [28], and we'll use an AWS ACM certificate for our domain. Internal traffic (ECS to RDS) stays within the VPC and can be encrypted (Postgres SSL). We'll document these encryption measures, as HITRUST will want to see that all PHI data is always encrypted both at rest and in transit (which it will be under this design).

Network Security & VPC Configuration – As mentioned, our VPC and subnet layout provides isolation. We will prepare a **network architecture diagram** showing the VPC, subnets, ALB, ECS, RDS, etc., as required for the HITRUST assessment [29]. The diagram will illustrate how data flows from the user's browser to our ALB, into ECS, to the database, and to Epic's API, etc., highlighting where PHI travels. This helps in the i1 assessment to show we've scoped and secured each data flow. We'll also highlight security controls like Security Groups and NACLs. Additionally, we'll implement an **AWS WAF** on the ALB to mitigate web threats (SQL injection, XSS, etc.) [15]. This adds another layer of defense and is often looked upon favorably in audits. We can use Managed Rule sets (including AWS's Core rule set or any healthcare-specific rules if available).

Monitoring, Alerting, Audit – AWS gives us CloudTrail, Config, and CloudWatch to meet many logging and monitoring controls. We will ensure CloudTrail is enabled for all regions and is logging to an S3 bucket (with retention) for forensic needs [11]. We'll also consider using **AWS Config** to continually evaluate our AWS resource configurations (for example, alert if any S3 bucket ever becomes public or if security groups become too open). Our audit logging in-app (e.g., AdminLog of user activity) is already implemented [23]. We will integrate these with AWS mechanisms: for instance, exporting AdminLog records periodically to a secure S3 or a DynamoDB for long-term retention, or enabling CloudWatch insights on our app logs to detect suspicious patterns. HITRUST i1 will require that we demonstrate *audit logging* and *incident monitoring* – by combining app-level logs with AWS logs, we'll have a comprehensive view. We will set up

CloudWatch Alarms or AWS Security Hub alerts for events like multiple failed logins (tie into our existing brute-force detection which is 10 attempts/5 min) [30], or any unauthorized AWS console login attempts, etc. These alerts will email the security team (and perhaps integrate with our existing Resend email alerts for security events [31]).

Data Retention & Backup – HITRUST requires proper data retention and disposal policies. Our plan is to use RDS backups and S3 to satisfy retention. For example, audit logs must be kept ~7 years [32] – we can achieve this by daily exporting of new audit log entries to an S3 "archive" bucket with a 7-year lifecycle. Similarly, our **Business Continuity/Disaster Recovery plan** (which we have documented) will be updated for AWS: we will note RDS backup retention (at least 7 days automated, plus long-term snapshots), **multi-AZ deployment** of RDS for high availability, and possibly cross-region backups for disaster recovery [33] [34]. We'll test our recovery by simulating a restore (as outlined in our testing recommendations) [35]. The ability to restore and re-sync from Epic is part of our deterministic recovery design [36] – AWS makes it easier to automate this, and we will include those procedures in our documentation.

HITRUST Documentation Updates – We will revisit all our security documents in the `docs/` folder to incorporate AWS: for example, the **HITRUST Readiness** checklist will be updated to mark network diagram as *implemented* once we have the VPC design [29]. Our **Key Management Policy** already includes AWS procedures (we'll follow those) [37]. The **Incident Response Plan** may need an update to include contacting AWS support if a breach involves AWS infrastructure. The **Business Continuity Plan** will include steps like promoting a read replica or restoring from RDS snapshot, etc. [38] [39]. We will also leverage AWS Artifact to download AWS compliance reports (HITRUST certification letters, SOC2 reports) which we can use as evidence for auditors showing AWS's controls.

Shared Responsibility Matrix Consideration – We will use the AWS HITRUST Shared Responsibility Matrix as a guide to ensure we're covering all controls. For example, if a control is about vulnerability scanning: AWS might cover scanning of their infrastructure, but we need to scan our application container and code. We'll continue our monthly dependency vulnerability scans and container image scans (perhaps using Amazon ECR's built-in scan or a tool like Trivy). Another example is access control: AWS will manage its employee access to infrastructure, but we manage our IAM users and application user access – so we'll enforce MFA on AWS accounts, principle of least privilege in IAM, and our application's RBAC (which is already implemented with roles) [40]. By aligning each HITRUST control with either AWS's responsibility or ours (or both), we can confidently meet the i1 assessment requirements.

Epic App Integration Considerations

Migrating hosting environment means our integration with **Epic (App Orchard)** must be adjusted but remain seamless. Key items to address:

- **Redirect URI**: Epic's OAuth will redirect users back to our application after authentication. We currently have a redirect URI registered (likely pointing to Replit URL or a test domain). Once on AWS, we'll have a new base URL (e.g., `https://app.healthprep.com/` or similar). We need to register a **new Redirect URI** with Epic that matches our production domain and OAuth callback route. In our Flask app, the OAuth callback route (perhaps in `routes/oauth_routes.py`) will handle the code exchange. We must ensure the domain in Epic's config exactly matches where the app will be deployed (including path, and using HTTPS). If we maintain a separate staging (sandbox) deployment for Epic's testing, that gets its own redirect URI in Epic's config as well.

- **Public Documentation URL**: Epic typically requires a URL with information about the application (for their review or for end-users to see details of the integration). We have implemented an endpoint `/epic/documentation` in the app [41] which likely serves a static page describing HealthPrep. This endpoint will need to be accessible without authentication. After migration, `https://**our-domain**/epic/documentation` will be the new public doc URL. We will update the App Orchard registration with this new URL. We'll double-check that the content on that page meets Epic's requirements (it may include things like a description of our app, support contact info, etc.). Since it's served from our app, we just need to keep that route public and functional.

- **JWKS (JSON Web Key Set) URLs**: Epic uses JWKS endpoints to fetch our public keys for verifying JWTs (if we use JWT client assertions for OAuth client auth). Our app already has robust JWKS endpoints for both non-production and production keys. Specifically, the Flask blueprint `epic_public_bp` provides:

- `/.well-known/jwks.json` for Production JWKS [42]
- `/nonprod/.well-known/jwks.json` for Non-Prod JWKS [43]. These need to be hosted at our new domain. The routes are defined, so as soon as the app is running on AWS and accessible via our domain, Epic will be able to hit, for example, `https://app.healthprep.com/.well-known/jwks.json` to get our keys. We must ensure that our ALB and Flask app allow these routes without auth (they should, as they're meant to be public). We will provide Epic the new URLs for the JWKS (both prod and sandbox) so they can update our App record.

Our key management policy indicates we name keys like `P_KEY_2025_08_A` (for production) and `NP_KEY_2025_08_A` (for non-prod). These are RSA private keys (PEMs) stored as env variables. The JWKS service in the app reads all env vars starting with `P_KEY` or `NP_KEY` and serves their public components in the JWKS response [44] [45]. We'll migrate these keys into AWS Secrets Manager (stored as PEM text) under the appropriate names, and ensure the ECS task exports them as env vars (likely via secretsmanager as well). As long as that's done, our JWKS endpoints will automatically publish the correct public keys. We should test this by hitting the JWKS URL after deployment to see that it returns a valid JSON with keys.

- **Epic OAuth Client Configuration**: Aside from redirect and JWKS, Epic App Orchard might require a **"Non-Production JWK Set URL"** and **"Production JWK Set URL"** separately (as the user noted). We have those covered by the two endpoints above – we'll just supply the appropriate one for each environment. Epic also often asks for a **Launch URI** or "FHIR base URL" if we were a SMART on FHIR app launched from within Epic, but in our case HealthPrep might act more as an external service that calls Epic's API. We should verify if Epic expects any other endpoints (the code also shows a `/health` endpoint likely used for a connectivity check [46] and a `/.well-known/smart-configuration` endpoint [47] which provides a standard config document for SMART on FHIR). We will ensure these are also reachable on the new domain. The `.well-known/smart-configuration` is likely needed by Epic to verify our capabilities (contains our auth URLs, etc.). After migration, we'll test that:
- `https://app.healthprep.com/.well-known/smart-configuration` returns the expected JSON,

- `https://app.healthprep.com/health` returns OK (for Epic's connectivity test).

- **Epic Key Rotation & Storage**: Our Epic client IDs and secrets, as well as the private keys for JWT, will be stored in AWS Secrets Manager (as mentioned). For instance, `EPIC_NONPROD_CLIENT_ID`, `EPIC_NONPROD_CLIENT_SECRET`, and `NP_KEY_2025_08_A` (private key for sandbox JWT) will go into a dev/test secrets path, while the production ones go into prod path [48] [49]. Post-migration, we will rotate these keys if needed by Epic's schedule. Epic typically issues client secrets or expects you to rotate keys periodically. According to our Key Management table, the Epic Prod key rotation is "Per Epic requirements" [50], so we'll handle that as needed (possibly generating a new RSA key annually or as required, updating Secrets Manager, and updating our JWKS – Epic would then fetch the new key via JWKS URL automatically). We should plan for a potential **additional redirect URI** if we have both a staging and prod environment accessible – the user's note suggests we might need to register another redirect (perhaps one for a dev environment on Replit and one for the new AWS prod). In any case, we'll coordinate these changes closely with Epic's support to avoid downtime (e.g., add the new redirect URI and JWKS in Epic config *before* switching over traffic to AWS).

- **Testing after Migration**: Once the app is live on AWS, we will test the full Epic OAuth flow in the **sandbox environment first**. Using Epic's launch or by simulating an OAuth token request, ensure that:

- Epic's authorization server can redirect to our new URI and our app successfully receives the auth code.
- Our app (or backend service) can exchange that code for a token from Epic – if we use JWT client assertion, confirm that Epic accepts our JWT (meaning they could fetch our JWKS and verify the signature).

- API calls to Epic (for pulling patient documents, etc.) work from the AWS environment (outbound internet access is set via NAT). After sandbox testing, we'll do a controlled test with production if possible (or during cutover, ensure one user can authenticate via production credentials). This will validate that all Epic integration points (redirect, JWKS, client IDs, etc.) are correctly configured post-migration.

- **Epic App Registration Update**: We will prepare a summary of changes to provide to the Epic App Orchard team or update via their portal: new domain for the app, new redirect URLs, JWKS URLs, and possibly a new **Public Base URL** if they have one on record. Since our application deals with PHI via Epic, Epic might also require confirmation that our new hosting environment is secure – our move to AWS (with BAA and HITRUST alignment) should satisfy that, and we can mention it if needed in our communication with them.

Key Management & Key Rotation on AWS

Secure key management is a cornerstone of both our application security and HITRUST compliance. We have a detailed **Key Management Policy** that covers rotation procedures for all secrets [37]. Here's how we'll implement it on AWS:

Migrating Secrets to AWS – All secrets from Replit's environment will be migrated into AWS Secrets Manager before deployment. The policy's mapping table shows where each secret goes [49]. For example: - `SESSION_SECRET` (Flask session cookie secret) → Secrets Manager at `/healthprep/prod/session-secret` [51] [52], - `ENCRYPTION_KEY` (Fernet key for DB field encryption) → `/healthprep/prod/`

`encryption-key`, - `P_KEY_2025_08_A` (Epic production RSA private key) → `/healthprep/prod/epic/prod-key`, - `NP_KEY_2025_08_A` → `/healthprep/dev/epic/nonprod-key`, etc.

We will use the AWS CLI or console to create these secrets and populate them with the values from Replit. In code, since we load via env vars, the ECS task definition will map these secret ARNs to env var names [8] so that, for instance, the value at `/healthprep/prod/secret-key` populates the `SECRET_KEY` variable at runtime. We'll be careful to place sandbox vs. prod secrets in the appropriate path (the app's JWKS logic expects non-prod keys with prefix `NP_KEY` in the dev/QA environment).

After migration, **no secrets will be stored in Git or in the container image** – they'll only reside in Secrets Manager (which is protected by KMS). This approach not only meets best practices but also specific HITRUST controls around key management (secure storage, limited access, audit of retrieval).

Using AWS for Key Rotation – Our policy dictates rotation frequencies (e.g., annual rotation for keys like SECRET_KEY, STRIPE keys, etc., or rotation per third-party requirements for Epic keys) [53]. On AWS, we have two methods to rotate: **manual (using AWS CLI/Console)** and **automated (for certain secrets)**.

- *Manual Rotation via AWS CLI:* The key management policy provides a step-by-step CLI procedure for rotating a standard secret [54] [55]. In summary, to rotate a secret like `SESSION_SECRET`, we would:
- Generate a new random value (for example, using Python's `secrets.token_hex(32)` for a 32-byte token) [56].
- Call `aws secretsmanager put-secret-value` to put the new value into the existing secret's latest version [57].
- Force the application to pick up the change – for ECS, that means triggering a new task deployment. We can do this with `aws ecs update-service --cluster healthprep-cluster --service healthprep-service --force-new-deployment` [58] (our cluster/service names will be whatever we set, "healthprep-cluster" is in the example). For a moment, two versions of the secret exist (old and new), but our app will only read the newest version on fresh tasks.
- Verify the new tasks are running and the application is functioning with the new secret (sessions might be invalidated if it's a session key, which is expected) – our policy notes to schedule such rotations in a maintenance window because session secrets invalidate user sessions [59].
- After verification, retire the old secret version if appropriate and document the rotation in our security log.

We will use a similar CLI approach for API keys (Stripe, Resend): update the secret, then redeploy or otherwise ensure the new key is in use, then disable the old key at the provider side [60] [61].

- *Automatic Rotation:* AWS Secrets Manager can auto-rotate certain secrets (notably database credentials) using Lambda functions. Our policy even gives an example of enabling rotation for the database URL (credentials) with a 30-day schedule [62]. If we switch to RDS and use a Secrets Manager-stored credential, we could enable this to automatically rotate the DB password and update the RDS user – however, since our app would need to pick up the new `DATABASE_URL`, we'd still need to restart tasks. It may be simpler to rotate DB credentials manually during planned maintenance. That said, we might enable the rotation with a Lambda that also triggers an ECS deploy (this would require some customization beyond the default Lambda).

For most application secrets, automation is tricky (because of needed coordination), so manual rotation is preferred. **Important**: The `ENCRYPTION_KEY` used for our DB field encryption **cannot** be auto-rotated by Lambda [63] because rotating it requires re-encrypting application data. We have a special process for that (discussed next).

- **Dual-Key Encryption Key Rotation:** Our PHI encryption key (ENCRYPTION_KEY) is critical and rotation involves re-encrypting all PHI in the database. The policy outlines a dual-key strategy [64] [65] :
- We generate a new encryption key (32-byte hex) and **add it as a secondary**: in AWS, we'd update the secret JSON for `/healthprep/prod/encryption-key` to hold both old and new keys (perhaps as a JSON with fields "current" and "pending") [66] . Or we could create a separate secret for the new key (`ENCRYPTION_KEY_NEW`) and have the app read both.
- Deploy an app update that can use two keys at once (our code has a `DualKeyEncryption` class ready for this, which tries to decrypt with the new key but falls back to old, and encrypts with new if available) [67] [68] . We would flip a feature flag or config to enable dual-key mode.
- Run a migration script (as provided in `scripts/reencrypt_phi_data.py`) to re-encrypt all PHI fields in the database with the new key [69] [70] . This will decrypt using the old key (the code tries new key first, then old) and immediately re-encrypt with the new key, updating each record. We'd do this in batches during a maintenance window (and we have backups taken before in case anything goes wrong) [71] .
- Once all data is re-encrypted, we promote the new key to be the primary (in Secrets Manager, that means update `/healthprep/prod/encryption-key` to just contain the new key as the current value). Then deploy the app back to single-key mode (i.e., remove the dual key code or just stop supplying the old key env var).
- Finally, securely destroy the old key material. In AWS, that means deleting the old secret version and any backups of it. Our policy also says to **not retain** old keys after rotation [72] .

We will carefully log and audit this process, as it's a significant cryptographic operation. The CLI steps for this (adding secondary key, etc.) are given in the policy and we will follow them. It's likely we'll perform this once as part of the migration (if we decide to rotate the encryption key when moving to AWS for extra security), or we'll schedule it at a later date if the current key isn't due for rotation yet.

- **Key Rotation Auditing** – Every time we rotate a secret on AWS, that action is recorded in CloudTrail (as a Secrets Manager `PutSecretValue` event). We will periodically run queries or use AWS CloudTrail Lake to ensure all rotations are logged and to check if any unauthorized secret changes occurred. The policy even suggests a CLI to lookup secret rotation events in CloudTrail [73] . During HITRUST assessment, we can show these logs to prove we follow our rotation schedules.

- **IAM Controls for Secrets** – We will create a specific **IAM role** for our ECS tasks that allows read access to only the necessary secrets (e.g., the app task role can get the values for `/healthprep/prod/*` secrets) and nothing more. Separately, an **IAM user or role for administrators** will have rights to rotate secrets. The policy provides an example IAM snippet showing least-privilege permissions to rotate secrets and trigger ECS deploys [74] [75] . We will implement something similar so that our CI/CD or ops team can run the AWS CLI rotation commands securely. This segregation ensures that even if the app is compromised, it cannot itself alter secrets – only read them – and rotations must be done by an authorized admin role.

- **Transfer of Existing Keys** – We will transfer existing keys (like the RSA keys for Epic) carefully. These are sensitive and likely stored as multiline PEM text. We will use AWS CLI or console to put these into Secrets Manager, making sure formatting is preserved. Once in AWS, we can enable secret value rotation for them if appropriate (though likely not automatic – we'll rotate via generating new key pairs when needed). The **Epic JWKS** system in our app is designed to handle multiple keys (it can list all keys with prefix) which will ease transitioning keys (we can introduce a new key alongside the old one and both will be exposed until we retire one, allowing key rollover without downtime) [76] [77].

In summary, AWS Secrets Manager will be the cornerstone of our key management on AWS. We will follow the documented procedures to rotate keys using AWS CLI for precision and scriptability, ensuring that for each rotation we update the secret, redeploy the app (ECS task refresh), verify functionality, and then disable old credentials [54] [55]. This approach keeps our application secrets fresh and minimizes the window of exposure, aligning with HITRUST CSF 10.g requirements for cryptographic key management [37].

CI/CD Pipeline (GitHub Actions to AWS ECS)

To streamline deployments to AWS, we will set up a **Continuous Integration/Continuous Deployment (CI/CD) pipeline** using GitHub Actions (since the code is hosted on GitHub and the team is familiar with it). This addresses point 4 (need for CI/CD suggestions) and ensures we can deploy changes reliably.

GitHub Actions Workflow – We'll create a workflow YAML in the repo (e.g., `.github/workflows/deploy.yml`). Key steps in this workflow:

1. **Trigger**: We can trigger on pushes to specific branches (for example, `main` for production deployments, and maybe a `dev` branch for a staging ECS service). For now, assume deploying to prod on push to `main` [78].

2. **Checkout Code**: Use `actions/checkout@v2` to pull the repository code in the CI runner [79].

3. **Set up AWS Credentials**: Use the AWS Actions to configure credentials [80]. We will create a GitHub Actions secret for `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and optionally `AWS_REGION`. These correspond to an IAM user or, better, an OpenID Connect federated role for GitHub (to avoid long-lived keys). Initially, using an IAM user with minimal permissions (ECR push, ECS deploy) is simplest. The `aws-actions/configure-aws-credentials@v1` action will export these creds in the environment, logging us into AWS within the workflow.

4. **Docker Build & Push**: The workflow will build the Docker image for the application. For example:

   ```
   docker build -t healthprep:${GITHUB_SHA} .
   ```

   (we tag with commit SHA or `latest`). Then log in to Amazon ECR and push the image. We can use AWS CLI for ECR login or the `aws-actions/amazon-ecr-login` action. The snippet in our docs simplifies by assuming $ECR_REGISTRY and using docker directly [81]. It would be something like:

```
aws ecr get-login-password --region us-east-1 | docker login --username
AWS --password-stdin <AWS_ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com
docker tag healthprep:${GITHUB_SHA} <AWS_ACCOUNT_ID>.dkr.ecr.us-
east-1.amazonaws.com/healthprep:latest
docker push <AWS_ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/
healthprep:latest
```

We'll set up the repository in ECR beforehand (named `healthprep`). The CI user/role needs permissions to perform ECR pushes.

5. **Deploy to ECS**: After pushing, we trigger the ECS service to deploy the new image. If we use the same image tag (e.g., `latest` or a version tag) in the task definition, we can force a refresh. The simplest approach (as shown in our docs) is:

```
aws ecs update-service --cluster <cluster-name> --service <service-name> --
force-new-deployment
```

[82]. This tells ECS to fetch the latest image and spin up new tasks with it. We might also use the Amazon ECS GitHub Action or AWS CodeDeploy for a blue/green deployment, but initially a rolling update is fine. The IAM permissions needed here are `ecs:UpdateService` (and the ability to describe services or task definitions if we script more). Our policy snippet already considered this [83].

6. **(Optional) Migrations**: If our deployment requires running a database migration or other setup (for example, using Flask-Migrate or Alembic), we would include a step to run that (perhaps by triggering a one-off ECS task or an AWS Lambda). This depends on how our app handles schema changes. We have a `scripts/` folder in the repo – if migrations are manual, we'll incorporate them into the process.

7. **Notifications**: We might add a step to notify (Slack/email) when a deployment succeeds or if it fails, so we have visibility. GitHub Actions can integrate with Slack or other tools.

This pipeline gives us a push-button (actually, push-triggered) deployment. For someone new to CI/CD, the key is that once it's configured, any code push will result in the app being updated on AWS within minutes – no manual FTP or UI changes needed. We will document this for the team and include instructions on how to roll back (for example, re-deploying a previous image tag if an issue is discovered).

CI Environment Security – We'll ensure the AWS credentials in GitHub are stored securely as GH Secrets. Also, limit the IAM user's permissions as mentioned (only ECR and ECS, not full admin). This mitigates risk if the CI pipeline is compromised.

Infrastructure as Code – As a side note, our Deployment Readiness doc *recommends using Terraform or AWS CDK to provision infrastructure*. We intend to follow that by writing Terraform scripts for creating the ECS cluster, task definitions, ALB, VPC, RDS, etc. The CI/CD pipeline discussed here is for application deployment (building the app image and updating the service). Infrastructure changes (like modifying a security group or scaling the database) would be managed via IaC and not happen on every app deploy. We might not fully automate Terraform in GitHub Actions yet, but at least maintain the config in source control.

CI/CD for Multi-Environment – If we maintain a **staging environment** on AWS (which is a good practice before deploying to prod), we can have separate ECS services (e.g., `healthprep-staging` and `healthprep-prod`). Our GitHub Actions can be configured with manual triggers or branch-based triggers. For example, a push to `develop` branch could deploy to the staging ECS, and only when we merge to `main` does it deploy to prod. This way we can test in staging (which might use Epic's sandbox credentials and JWKS nonprod endpoints) without affecting production. This aligns with the separation of `dev` and `prod` secrets paths in Secrets Manager [84] .

Closing the Loop – After CI/CD is in place, every code change will automatically go through the pipeline. This reduces human error and ensures consistency. We will monitor the first few deployments to make sure everything works (image builds are successful, ECS tasks register as healthy behind ALB, etc.). We'll also implement health checks in ECS (the `/health` endpoint of our app can serve for that) so ECS knows if a new task is healthy. The pipeline will only update the service; ECS handles rolling replacement of containers. If a deployment fails (container can't start or health check fails), ECS will stop deploying and we can investigate (and our CI can alert us). This gives us confidence to iterate quickly while remaining secure.

Conclusion

Preparing for the AWS migration involves addressing both **technical mapping** and **compliance requirements**. We have mapped each component of our system to an AWS service that best fits – ECS Fargate for compute, RDS for the database, S3 for file storage, Secrets Manager for secrets, and so on – and designed a secure VPC architecture to contain them [85] . This not only replicates our current functionality on a more robust platform, but also sets us up to meet **HITRUST i1** compliance by leveraging AWS's certified infrastructure and implementing required controls (encryption, audit logging, backup, etc.). We will create the necessary documentation artifacts (network diagrams, revised policies) to reflect the new AWS deployment and the shared responsibility considerations.

Crucially, we will update our Epic App Orchard configuration so that our integration continues to function post-migration – new redirect URIs, JWKS endpoints, and documentation URLs will be provided and tested to ensure Epic can seamlessly connect to HealthPrep on AWS. By analyzing the Key Management Policy, we've planned out how to use AWS's tooling (Secrets Manager, KMS, CloudTrail) to enforce strong secret handling and rotation, using CLI scripts to rotate keys without downtime and with auditability [54] [55] .

Finally, implementing a CI/CD pipeline via GitHub Actions will automate our deployment process to AWS ECS [82] . This means faster, more reliable releases and the ability to roll out security patches or updates quickly – a DevOps practice that compliments our security goals (e.g., timely patching is a HITRUST control as well).

By following this plan, we can achieve a smooth transition from Replit/GCP to AWS, improve our control over the environment, and be well-prepared for the HITRUST i1 assessment. The end result will be a containerized, scalable application on AWS that maintains the privacy-first architecture (PHI redaction, minimal storage) [19] we have built, with the additional safeguards and documentation needed for compliance. We'll proceed step by step – setting up AWS infrastructure, migrating data and secrets, updating configurations, testing thoroughly (both functionality and security), and then going live on AWS with confidence.

Sources:

- HealthPrep Deployment Readiness & Security Docs [15] [8] [85] [86]
- HealthPrep Key Management Policy [54] [87]
- HealthPrep Epic Integration Code (public endpoints for Epic) [41] [43]
- HealthPrep Security Whitepaper & HITRUST Readiness Checklist [3] [11] [29]

---

[1] [2] [4] [8] [12] [13] [14] [15] [16] [17] [18] [21] [22] [25] [27] [28] [35] [78] [79] [80] [81] [82] [85] [86]
DEPLOYMENT_READINESS.md
https://github.com/mitfusco98/HealthPrepV2/blob/f1d9e2d687653e0767a588ccbcc143ace2e25b71/DEPLOYMENT_READINESS.md

[3] [5] [6] [9] [10] [11] [19] [20] [26] [32] [33] [34] [36] [38] [39] SECURITY_WHITEPAPER.md
https://github.com/mitfusco98/HealthPrepV2/blob/f1d9e2d687653e0767a588ccbcc143ace2e25b71/docs/security/SECURITY_WHITEPAPER.md

[7] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] [73] [74] [75] [83] [84] [87] key-management-policy.md
https://github.com/mitfusco98/HealthPrepV2/blob/f1d9e2d687653e0767a588ccbcc143ace2e25b71/docs/security/key-management-policy.md

[23] [24] [29] [30] [31] [37] [40] HITRUST_READINESS.md
https://github.com/mitfusco98/HealthPrepV2/blob/f1d9e2d687653e0767a588ccbcc143ace2e25b71/docs/security/HITRUST_READINESS.md

[41] [42] [43] [46] [47] epic_public_routes.py
https://github.com/mitfusco98/HealthPrepV2/blob/f1d9e2d687653e0767a588ccbcc143ace2e25b71/routes/epic_public_routes.py

[44] [45] [76] [77] jwks_service.py
https://github.com/mitfusco98/HealthPrepV2/blob/f1d9e2d687653e0767a588ccbcc143ace2e25b71/services/jwks_service.py